系統晶片設計
SOC Design

LAB4-2

組員:郭銘宸、郭紘碩

學號:311511082、3115120656

系所:電機所碩二、電控所碩二

一、 Design Block Diagram

➢ 架構說明:

(1) WB decoder:

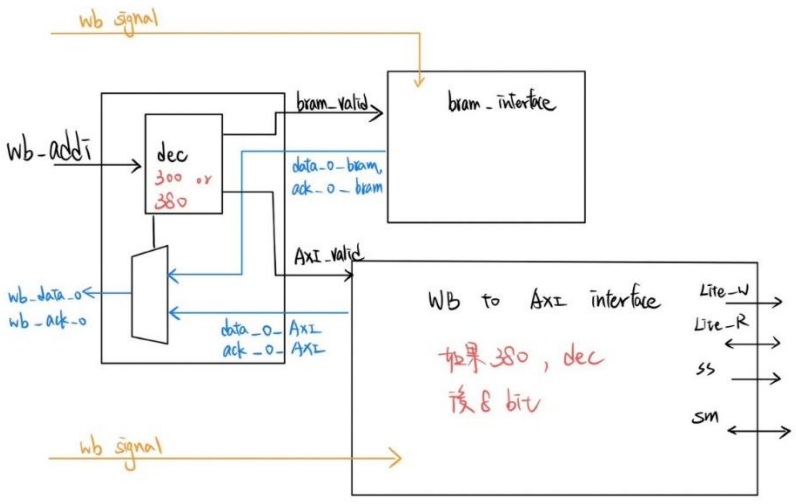
我 WB decoder 負責兩件工作，第一:decoder WB address 前 12bit，判斷 bram interface 或 axi interface 哪一個要啟動(380->bram，300->AXI)。第二:決定要讓哪一邊 output data 與 ack 送到 WB。

(2) WB to AXI interface

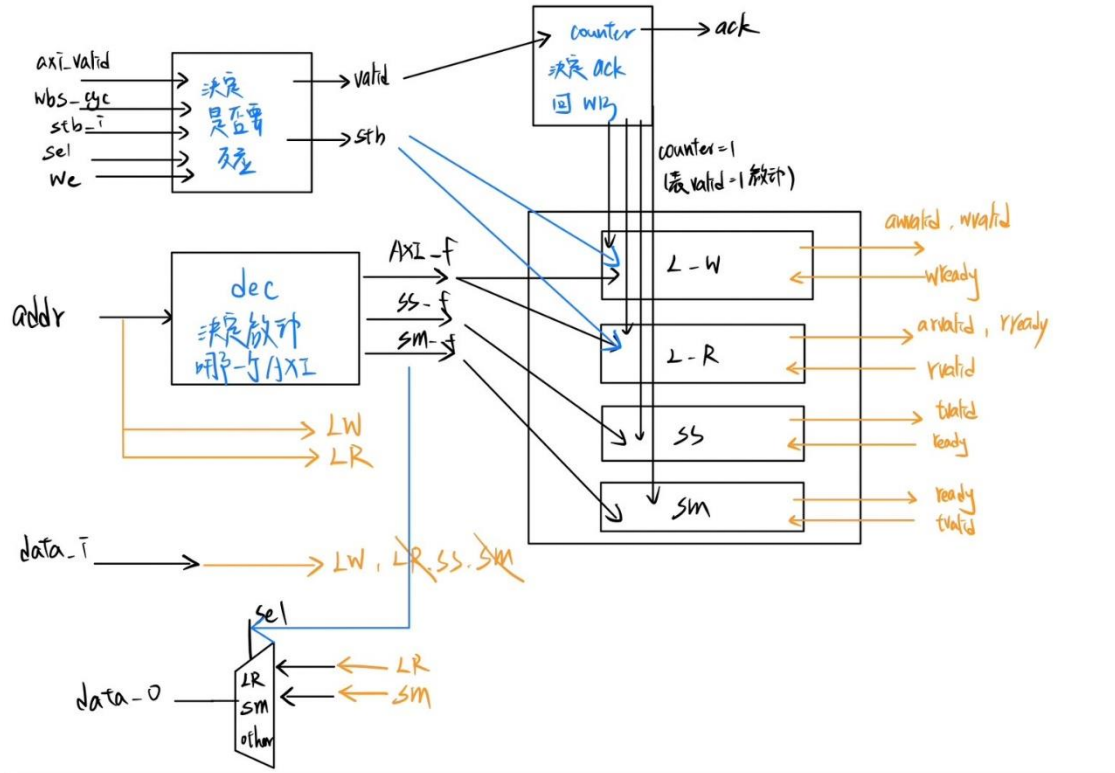
WB to AXI interface 主要負責三件工作，第一:判輸入 cyc、stb 與 WB decoder 送過來的 axi_valid 是否都是 1，都是 1 才要對這次 WB 訊號作反應。第二:當確定此 interface 要對 wb 反應後，decode 地址後 8bit，因為前面 3x0 已經解碼過，所以只要對後面 8bit 解碼即可。第三:用 counter 決定何時回復 ack 給 WB。

(3) 加入 X、Y flag in lab3

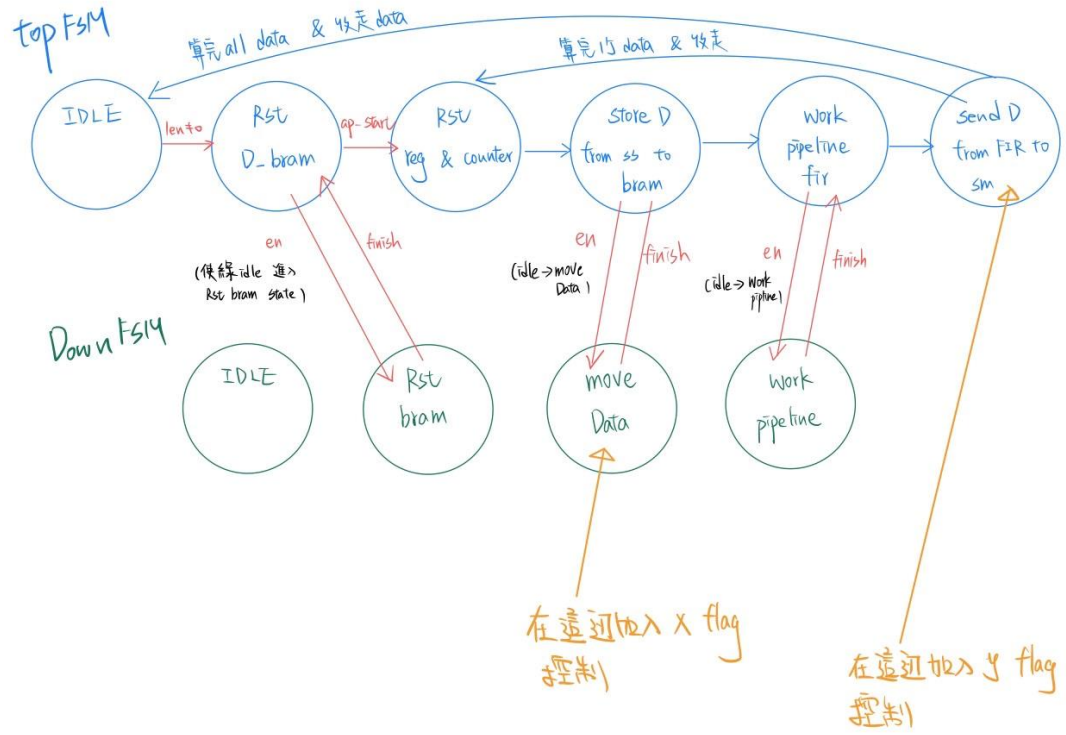
這次需要加入 X、Y flag，好讓 WB 知道 fir 是否可以收 X[n]與是否有計算出 Y[n]。依照我 lab3 的 FSM，當 store data from ss to bram state 時，會將 X flag reg 拉為 1，等到收完 ss data 會把 X flag reg 設為 0 表示不能收 data。當 send data from FIR to sm state 時，會將 Y flag reg 拉為 1，等待資料被收走後將 Y flag reg 設回 0，老師在 workbook 建議的是 read 0x00 時將 Y flag reg 設回 0，我這邊將其調整為收走就拉回 0(因為那頁投影片寫 suggested，所以我作一些調整)。



圖(1) WB decoder



圖(2) WB to AXI interface

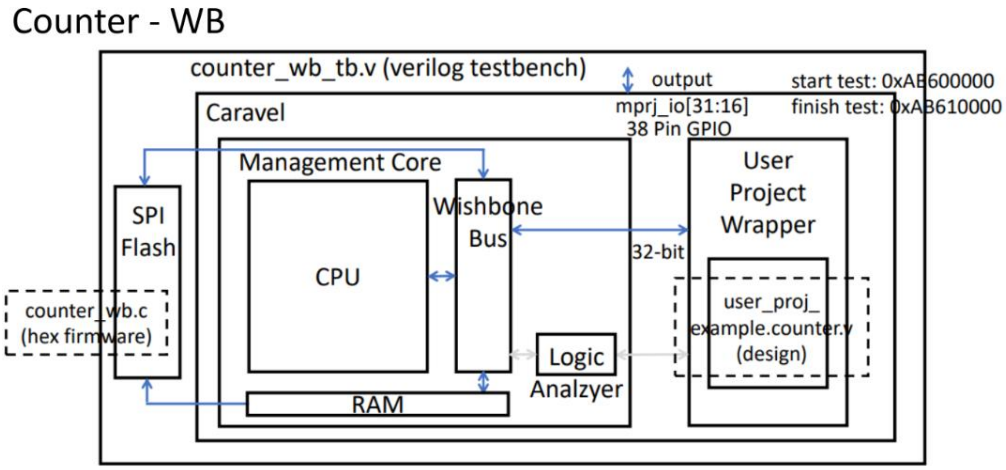


圖(3) 加入 X、Y flag in lab3

二、 The interface protocol between firmware, user project and testbench

➢ 說明

(1) Testbench 會將訊號打在 caravel 上。左圖整個 caravel 包含了 RISC-V CPU 與 User project wrapper，如下圖(4)所示。可以透過對 mprj 的設置(放在 counter_la_fir.c)，將 mprj 設定為 management core 的輸出，這樣 Testbench 可以透過 mprj 知道內部運算的值，下圖(5)所示。



圖(4) 模擬架構圖

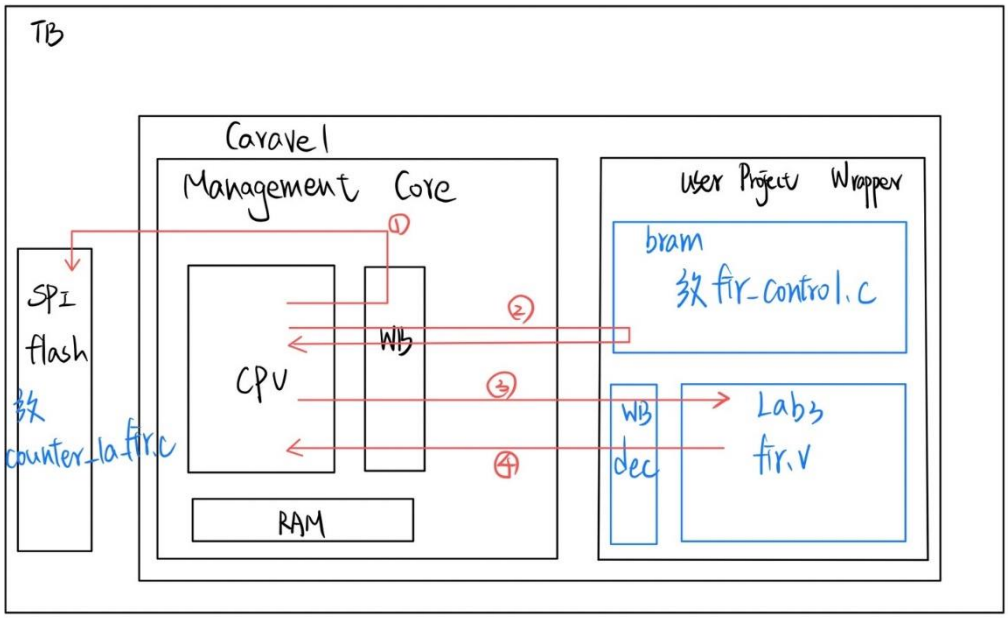
```
    reg_mprj_io_31 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_30 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_29 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_28 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_27 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_26 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_25 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_24 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_23 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_22 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_21 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_20 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_19 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_18 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_17 = GPIO_MODE_MGMT_STD_OUTPUT;
    reg_mprj_io_16 = GPIO_MODE_MGMT_STD_OUTPUT;

    reg_mprj_io_15 = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_14 = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_13 = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_12 = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_11 = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_10 = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_9  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_8  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_7  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_5  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_4  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_3  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_2  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_1  = GPIO_MODE_USER_STD_OUTPUT;
    reg_mprj_io_0  = GPIO_MODE_USER_STD_OUTPUT;

    reg_mprj_io_6  = GPIO_MODE_MGMT_STD_OUTPUT;

// Set UART clock to 64 kbaud (enable before I/O configuration)
// reg_uart_clkdiv = 625;
reg_uart_enable = 1;

// Now, apply the configuration
reg_mprj_xfer = 1;
while (reg_mprj_xfer == 1);
```

圖(5) 設定 mprj 的 firmware

(2) 剛開始模擬時 CPU 會發 0x1000000 的 wishbone cycle 到 SPI flash 拿 firmware code 執行，當執行到要送資料給 fir 硬體時，會發 wishbone cycle 到 bram 拿要跑的 firmware code(fir_control.c)來 CPU 執行，所以這時 wishbone address 為 3800000。

(3) 上面對 bram 拿 firmware 發的 3800000 也是對 user_project 發送，送資料到 fir 硬體的 30000000 也是對 user_project 發送，所以需要對地址解碼，詳細說明可看上面第一部份的(1)(2)。

(4) 整體 CPU 執行順序如(aka. WB cycle 發送順序)下圖(6)所示

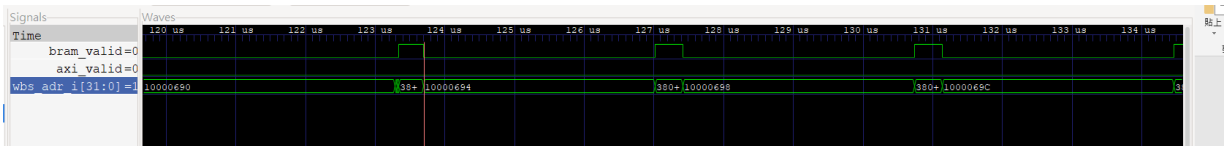


圖(6) CPU 執行順序(aka. WB cycle 發送順序)

三、 Waveform and analysis of the hardware/software behavior

➢ 放 fir_control.c 到 bram

當開始模擬時，前面會先將 fir_control.c 的 firmware code 放到 bram，可以看到下面圖(7)波形圖，地址是發 38000000，所以 decoder 的 bram_valid 會為 1。



圖(7) 放 fir_control.c 到 bram 波形圖

- Fir_control.c 與硬體溝通說明

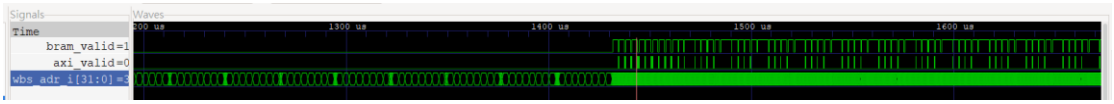
(1) 可以看到下圖(8-1)，這邊是開始執行 bram 裡面的 firware code，然後這個 firmware 工作為發 data 給 fir 硬體，所以會很常切換 380000 與 300000。

(2) 可以看到下圖(8-2)，第一次發 3000000 是為了讀取是否 fir 硬體內的 ap_idle 為 1。從標記線可以看到，當有效 30000000 時(cyc 與 stb 同時為 1)，axi_valid 為 1，然後使 axi interface 內的 valid 為 1，啟動 axi 送資料給 fir 硬體，可以看到下面啟動 axi_lite 的 read protocol。

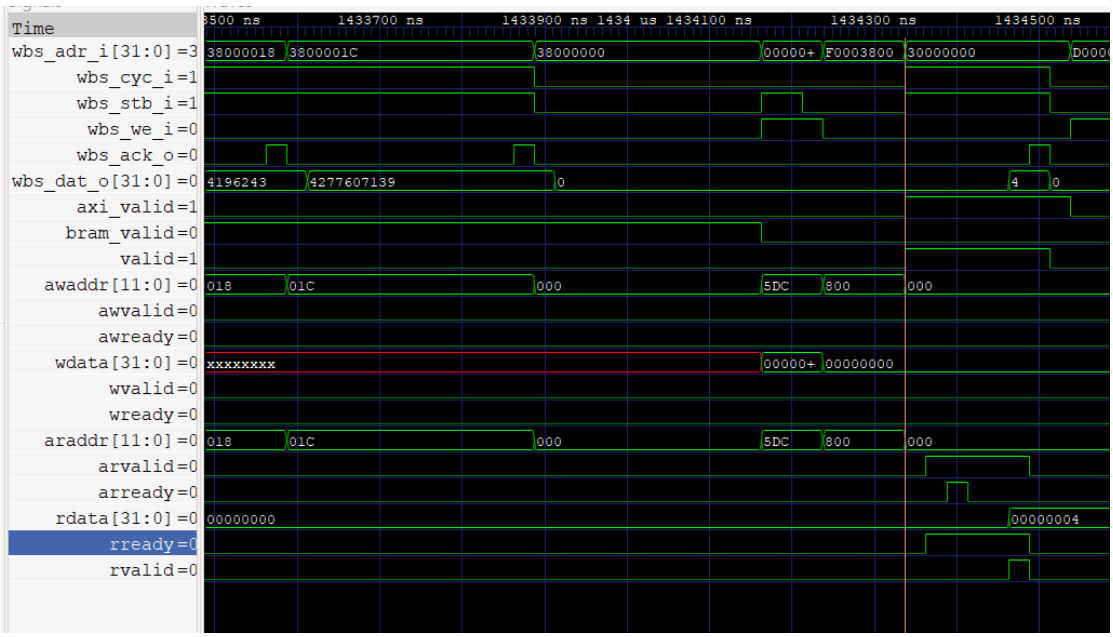
(3) 可以看到下圖(8-3)，再來會執行送 length(30000010)與 tap(30000040~30000068)，再來是將 ap_start 設為 1，這邊都是啟動 axi_lite 的 write 的 protocol。

(4) 可以看到下圖(8-4)，啟動 fir 後，每次寫 X 進去 fir 都要先讀到 30000000 的 data 為 16(也就是 fir 的 X flag 為 1)，才會送 X 進去 fir。在由圖(8-5)可以看到，接續讀到 X flag 為 1 後，收到 30000080 啟動 stream slave protocol。

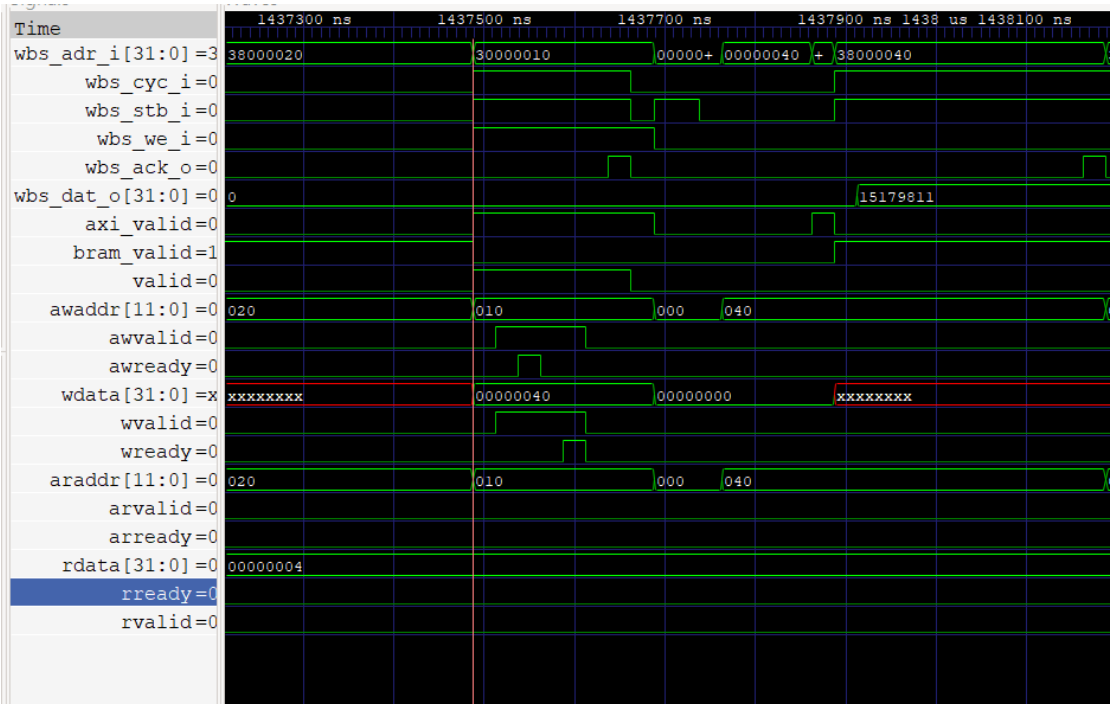
(5) 可以看到下圖(8-6)，每次讀取 fir 運算結果時會先去讀 30000000 的 data 為 32(也就是 fir 的 Y flag 為 1)，才會讀取 fir 運算結果。在由圖(8-7)可以看到，接續讀到 Y flag 為 1 後，收到 30000084 啟動 stream master protocol。這邊可以看到 sm_tvalid 在 fir 算完後會一直拉 1 等待 axi interface 發起 ready 收走此筆資料。



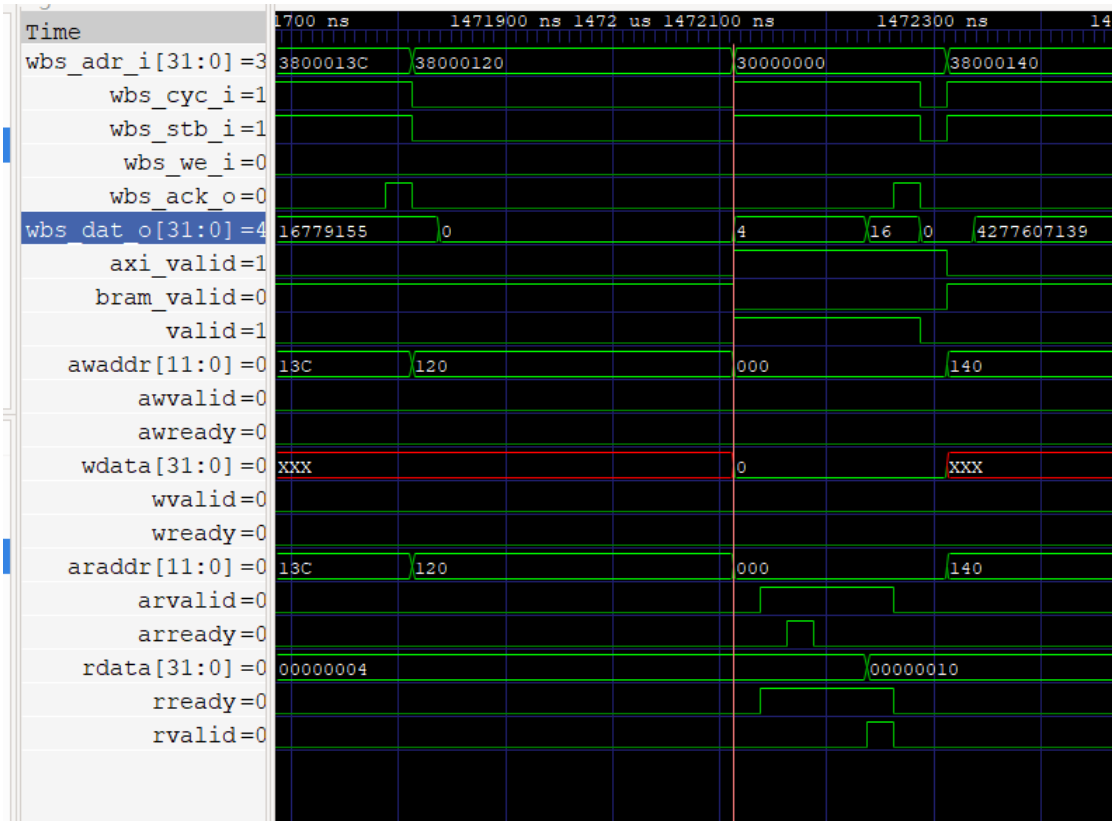
圖(8-1)



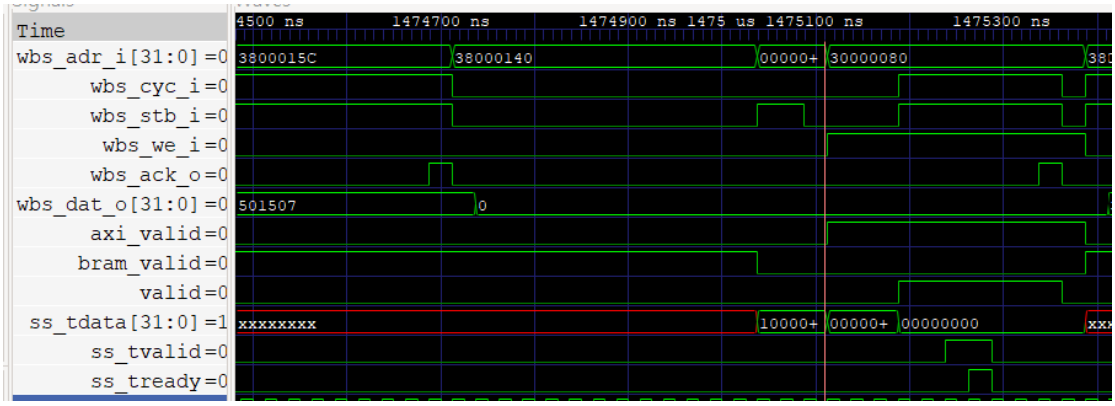
圖(8-2) WB 轉 AXI lite 的 read



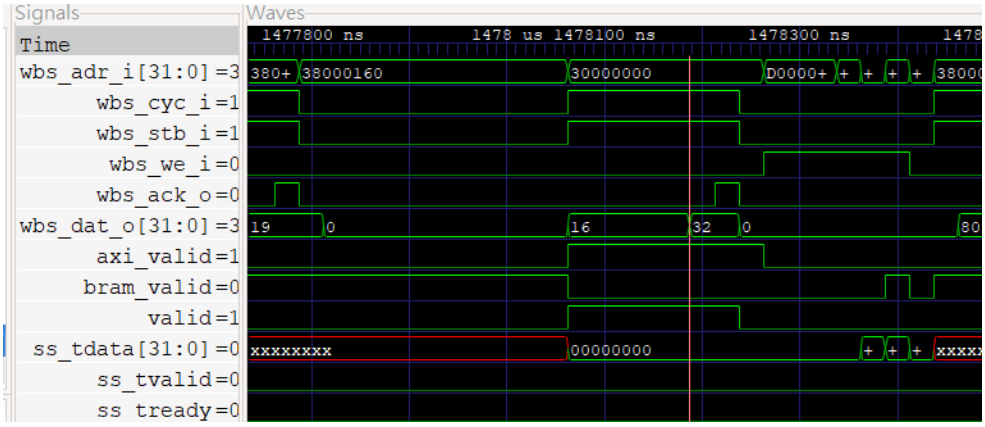
圖(8-3) WB 轉 AXI lite 的 write



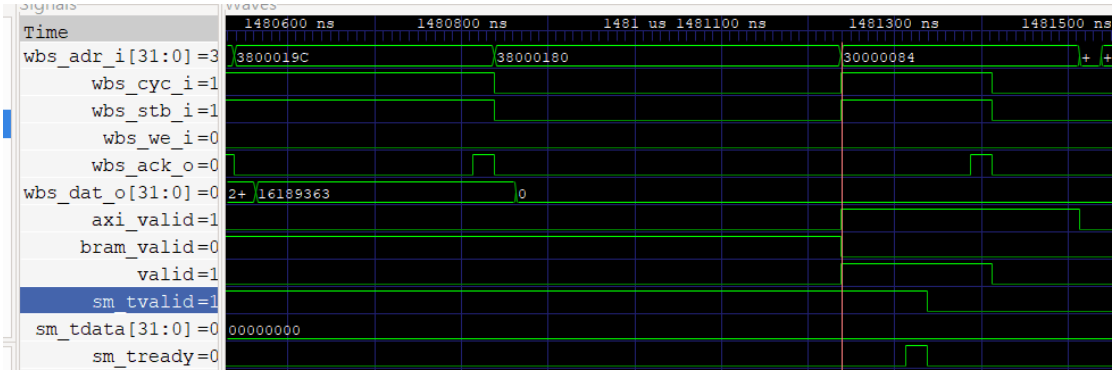
圖(8-4) 讀 X flag



圖(8-5) WB 轉 SS



圖(8-6) 讀 Y flag



圖(8-7) WB 轉 SM

四、 實驗結果截圖

- 跑三次+每次的 time

```
ubuntu@ubuntu2004:~/course/lab4_2/testbench/counter_la_fir$ source run_sim
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started
t_start Elapsed time:    1461088 [ns]
t_end Elapsed time:    2634638 [ns]
Elapsed time:    1173550 [ns]
0x76 mprj: data = 0x76
LA Test 1 passed
LA Test 2 started
t_start Elapsed time:    2697888 [ns]
t_end Elapsed time:    3871438 [ns]
Elapsed time:    1173550 [ns]
0x76 mprj: data = 0x76
LA Test 2 passed
LA Test 3 started
t_start Elapsed time:    3961113 [ns]
t_end Elapsed time:    5134663 [ns]
Elapsed time:    1173550 [ns]
0x76 mprj: data = 0x76
LA Test 3 passed
ubuntu@ubuntu2004:~/course/lab4_2/testbench/counter_la_fir$
```

圖(9) 跑三次模擬圖

- Resource usage: 包含 WBdec.v、WBtoAXI.v、fir.v、bram11.v、user_proj_example.counter.v、bram.v

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	632	0	0	53200	1.19
LUT as Logic	504	0	0	53200	0.95
LUT as Memory	128	0	0	17400	0.74
LUT as Distributed RAM	128	0			
LUT as Shift Register	0	0			
Slice Registers	542	0	0	106400	0.51
Register as Flip Flop	363	0	0	106400	0.34
Register as Latch	179	0	0	106400	0.17
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
28	Yes	-	Set
489	Yes	-	Reset
0	Yes	Set	-
25	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	1	0	0	140	0.71
RAMB36/FIFO*	1	0	0	140	0.71
RAMB36E1 only	1				
RAMB18	0	0	0	280	0.00

圖(10) synth report

- Timing report

說明:可以看到最長 critical path 在 fir 裡面，需要 11.6ns，故 cycle period 設 11ns 可以通過 slack。

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.223ns  (required time - arrival time)
  Source:                 fir_top/fir_data_reg_out_reg[16]/C
                            (rising edge-triggered cell FDCE clocked by wb_clk_i  {rise@0.000ns fall@5.500ns period=11.000ns})
  Destination:            fir_top/fir_result_out_reg_out_reg[29]/D
                            (rising edge-triggered cell FDCE clocked by wb_clk_i  {rise@0.000ns fall@5.500ns period=11.000ns})
  Path Group:             wb_clk_i
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            11.000ns  (wb_clk_i rise@11.000ns - wb_clk_i rise@0.000ns)
  Data Path Delay:        10.672ns  (logic 8.380ns (78.520%)  route 2.292ns (21.480%))
  Logic Levels:           9  (CARRY4=5 DSP48E1=2 LUT2=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.128ns = ( 13.128 - 11.000 )
    Source Clock Delay      (SCD):    2.456ns
    Clock Pessimism Removal (CPR):    0.184ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

圖(11) timing report

五、 What is the FIR engine theoretical throughput? Actually measured throughput?

- 說明:

(1) Throughput -> 單位時間內可以處理的 operate(所以應該是 ap_start 設為 1 到 ap_idle 為 1 做完)

(2) theoretical throughput -> 以 lab3 為範例，因為它的輸入是一直傳來。

(3) Actually measured throughput -> 則為 lab4-2，需要使用 CPU 傳輸資料的

- theoretical throughput(lab3)(25ns)

Operations =11*600*2

Latency = 13804c.c * 25ns =345100ns

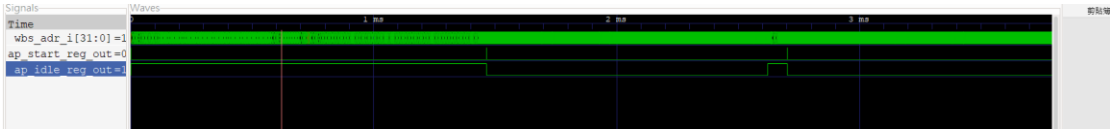
Throughput = 38249782.67 ops/s

- Actually measured throughput(lab4-2)(clock period = 25ns)

Operations=11*64*2

Latency = (2620762500-1463937500)ps

Throughput = 1217124.457 ops/s



圖(12) lab4-2 ap_start=1 to ap_idle=1

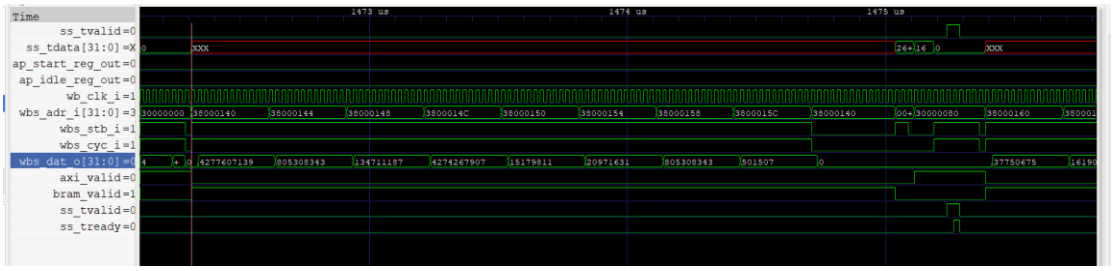
六、 What is latency for firmware to feed data?

- 說明:

(1) firmware to feed data 的 latency -> 起始時間是 CPU 收到 X flag 為 1 後到 380000 去拿要送 X 給 fir 的 firmware，結束時間為 WB decoder 收到 30000080。如下圖(13)所示，標記線為起始時間，300080 是結束。

(2) Latency = (1475112500-1472312500)ps = 2800000ps

2800000/25000 = 112 clock cycle



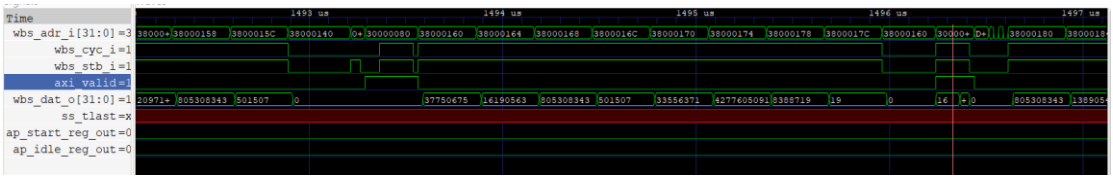
圖(13)

七、 What techniques used to improve the throughput?

- Does bram12 give better performance, in what way?

可以提升 performance。可以透過圖(14-1)可以看到標記線之處是我去判斷 Y flag 是否為 1，因為我的 firmware 寫法是使用 while{}(圖(14-2))所以可以看到前面要跑很多 firmware code 才發出 WB cycle。會這樣寫是因為需要 firmware 等待 fir 拉起 Y flag。

先前的 firmware 我是用 if()(a.k.a 圖(14-3))，這樣會更快發起 WB cycle 但是發起時可能 fir 還不能收資料，導致 if(){}裡面的 x 沒送出去。若多一個 buffer 就可以用 if(){}因為有多一個 buffer，X flag 就不用等 fir 算完再拉起，CPU 可以更快的送 data。



圖(14-1) CPU 判斷 Y flag

```
for(int n=0; n<data_length; n++){

    while(reg_fir_control != 0x00000010) {}
    reg_fir_x = n;
    while(reg_fir_control != 0x00000020) {
        if(reg_fir_control == 0x00000024) {
            break;
        }
    }
    y[n] = reg_fir_y;
}
reg_mprj_datal = (0x005A00 | (y[63]<<16)) << 8;
```

圖(14-2) firmware while 版本

```
if(reg_fir_control == 0x00000010) {
    reg_fir_x = n;
}
```

圖(14-3) firmware if 版本

- Can you suggest other method to improve the performance?

可以透過增加 Buffer 來使 firmware 更快丟資料與收資料，不用使用 while 而是使用 if，然後可以增加多套一點硬體，可以增加資料算完速度也可以加快 buffer 消耗，以幣面塞車。

八、 Any other insights?

- Firmware 放 SPIflash 與 Bram 差異

  一開始我們是將 firmware 全部放在 SPIflash 跑出來的時間是 9200000 多，然後將 firmware 放到 bram 後馬上降到 1170000，如圖(9)所示。